Modular tree automata

Bahr, Patrick

Published in:
Mathematics of Program Construction

DOI:
10.1007/978-3-642-31113-0_14

Publication date:
2012

Document version
Peer reviewed version

Modular Tree Automata

Patrick Bahr

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Abstract. Tree automata are traditionally used to study properties of tree languages and tree transformations. In this paper, we consider tree automata as the basis for modular and extensible recursion schemes. We show, using well-known techniques, how to derive from standard tree automata highly modular recursion schemes. Functions that are defined in terms of these recursion schemes can be combined, reused and transformed in many ways. This flexibility facilitates the specification of complex transformations in a concise manner, which is illustrated with a number of examples.

1 Introduction

Functional programming languages are an excellent tool for specifying abstract syntax trees (ASTs) and defining syntax-directed transformations on them: algebraic data types provide a compact notation for both defining types of ASTs as well as constructing and manipulating ASTs. As a complement to that, recursively defined functions on algebraic data types allow us to traverse ASTs defined by algebraic data types.

For example, writing an evaluation function for a small expression language is easily achieved in Haskell [19] as follows:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Exp} = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp}\\
&\mathit{eval} :: \mathit{Exp} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ i) \quad = i\\
&\mathit{eval}\ (\mathit{Plus}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}
$$

Unfortunately, this simple approach does not scale very well. As soon as we have to implement more complex transformations that work on more than just a few types of ASTs, simple recursive function definitions become too inflexible and complicated.

Specifying and implementing such transformations is an everyday issue for compiler construction and thus has prompted a lot of research in this area. One notable approach to address both sides is the use of attribute grammars [15, 22]. These systems facilitate compact specification and efficient implementation of syntax-directed transformations.

In this paper, we take a different but not unrelated approach. We still want to implement the transformations in a functional language. But instead of writing transformation functions as general recursive functions as the one above, our goal is to devise *recursion schemes*, which can then be used to define the desired transformations. The use of these recursion schemes will allow us reuse, combine and reshape the syntax-directed transformations that we write. In addition, the embedding into a functional language will give us a lot of flexibility and expressive power such as a powerful type system and generic programming techniques.

As a starting point for our recursion schemes we consider various kinds of tree automata [3]. For each such kind we show how to implement them in Haskell. From the resulting recursion schemes we then derive more sophisticated and highly modular recursion schemes. In particular, our contributions are the following:

- We implement bottom-up tree acceptors (Section 2), bottom-up tree transducers (Section 4) and top-down tree transducers (Section 5) as recursion schemes in Haskell. While the implementation of the first two is well-known, the implementation of the last one is new but entirely straightforward.
- From the thus obtained recursion schemes, we derive more modular variants (Section 3) using a variation of the well-know product automaton construction (Section 3.1) and Swierstra's *data types à la carte* [23] (Section 3.2).
- We decompose the recursion schemes derived from bottom-up and from top-down tree transducer into a homomorphism part and a state transition part (Section 4.5 and Section 5.3). This makes it possible to specify these two parts independently and to modify and combine them in a flexible manner.
- We derive a recursion scheme that combines both bottom-up and top-down state propagation (Section 6).
- We illustrate the merit of our recursion schemes by a running example in which we develop a simple compiler for a simple expression language. Utilising the modularity of our approach, we extend the expression language throughout the paper in order to show how the more advanced recursion schemes help us in devising an increasingly more complex compiler. In addition to that, the high degree of modularity of our approach not only simplifies the construction of the compiler but also allows us to reuse earlier iterations of the compiler.

Apart from the abovementioned running example, we also include a number of independent examples illustrating the mechanics of the presented tree automata.

The remainder of this paper is structured as follows: we start in Section 2 with bottom-up tree acceptors and their implementation in Haskell. In Section 3, we introduce two dimensions of modularity that can be exploited in the recursion scheme obtained from bottom-up tree acceptors. In Section 4, we will turn to bottom-up tree transducers, which, based on a state that is propagated upwards, perform a transformation of an input term to an output term. In Section 4.5 we will then introduce yet another dimension of modularity by separating the state propagation in tree transducers from the tree transformation. This will also allow

us to adopt the modularity techniques from Section 3. In Section 5, we will do the same thing again, however, for top-down tree transducers in which the state is propagated top-down rather than bottom-up. Finally, in Section 6, we will combine both bottom-up and top-down state transitions.

The library of recursion schemes that we develop in this paper is available as part of the `compdata` package [2]. Additionally, this paper is written as a literate Haskell file[1], which can be directly loaded into the GHCi Haskell interpreter.

2 Bottom-Up Tree Acceptors

The tree automata that we consider in this paper operate on terms over some signature $\mathcal{F}$. In the setting of tree automata, a signature $\mathcal{F}$ is simply a set of function symbols with a fixed arity and we write $f/n \in \mathcal{F}$ to indicate that f is a function symbol in $\mathcal{F}$ of arity n. Given a signature $\mathcal{F}$ and some set $\mathcal{X}$, the set of terms over $\mathcal{F}$ and $\mathcal{X}$, denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is the smallest set T such that $\mathcal{X} \subseteq T$ and if $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in T$ then $f(t_1, \ldots, t_n) \in T$. Instead of $\mathcal{T}(\mathcal{F}, \emptyset)$ we also write $\mathcal{T}(\mathcal{F})$ and call elements of $\mathcal{T}(\mathcal{F})$ terms over $\mathcal{F}$. Tree automata run on terms in $\mathcal{T}(\mathcal{F})$.

Each of the tree automata that we describe in this paper consists at least of a finite set Q of states and a set of rules according to which an input term is transformed into an output term. While performing such a transformation, these automata maintain state information, which is stored in the intermediate results of the transformation. To this end each state $q \in Q$ is considered as a unary function symbol and a subterm t is annotated with state q by writing $q(t)$. For example, $f(q_0(a), q_1(b))$ represents the term $f(a, b)$, where the two subterms a and b are annotated with states q_0 and q_1, respectively.

The rules of the tree automata in this paper will all be of the form $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}', \mathcal{X})$, where $\mathcal{F}' = \mathcal{F} \uplus \{q/1 \mid q \in Q\}$. The rules can be read as term rewrite rules, i.e. the variables in l and t are placeholders that are instantiated with terms when the rule is applied. Running an automaton is then simply a matter of applying these term rewrite rules to a term. The different kinds of tree automata only differ in the set of rules they allow.

2.1 Deterministic Bottom-Up Tree Acceptors

A *deterministic bottom-up tree acceptor* (DUTA) over a signature $\mathcal{F}$ consists of a (finite) set of states Q, a set of accepting states $Q_a \subseteq Q$, and a set of transition rules of the form

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)), \quad \text{with } f/n \in \mathcal{F} \text{ and } q, q_1, \ldots, q_n \in Q$$

The variable symbols $x_1, \ldots, x_n$ serve as placeholders in these rules and states in Q are considered as function symbols of arity 1. The set of transition rules must be deterministic – i.e. there are no two different rules with the same left-hand

[1] Available from the author's web site.

side – and complete – i.e. for each $f/n \in \mathcal{F}$ and $q_1, \ldots, q_n \in Q$, there is a rule with the left-hand side $f(q_1(x_1), \ldots, q_n(x_n))$. The state q on the right-hand side of the transition rule is also called the *successor state* of the transition.

By repeatedly applying the transition rules to a term t over $\mathcal{F}$, initial states are created at the leaves which then get propagated upwards through function symbols. Eventually, we obtain a final state q_f at the root of the term. That is, an input term t is transformed into $q_f(t)$. The term t is *accepted* by the DUTA iff $q_f \in Q_a$. In this way, a DUTA defines a term language.

Example 1. Consider the signature $\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0\}$ and the DUTA over $\mathcal{F}$ with $Q = \{q_0, q_1\}$, $Q_a = \{q_1\}$ and the following transition rules:

$$\begin{array}{lll}
\mathsf{ff} \to q_0(\mathsf{ff}) & \mathsf{not}(q_0(x)) \to q_1(\mathsf{not}(x)) & \mathsf{and}(q_0(x), q_1(y)) \to q_0(\mathsf{and}(x, y)) \\
\mathsf{tt} \to q_1(\mathsf{tt}) & \mathsf{not}(q_1(x)) \to q_0(\mathsf{not}(x)) & \mathsf{and}(q_1(x), q_0(y)) \to q_0(\mathsf{and}(x, y)) \\
\multicolumn{2}{c}{\mathsf{and}(q_1(x), q_1(y)) \to q_1(\mathsf{and}(x, y))} & \mathsf{and}(q_0(x), q_0(y)) \to q_0(\mathsf{and}(x, y))
\end{array}$$

Terms over signature $\mathcal{F}$ are Boolean expressions and the automaton accepts such an expression iff it evaluates to true.

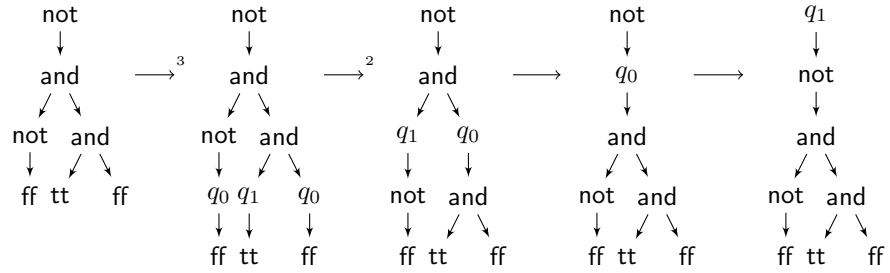
Note that the rules are complete – for each function symbol, every combination of input states occurs in the left-hand side of some rule – and deterministic – there are no two rules with the same left-hand side.

The transition rules are applied by interpreting them as rules in a term rewriting system, where variables are placeholders for terms. For the term $\mathsf{and}(\mathsf{tt}, \mathsf{ff})$, we get the following derivation:

$$\mathsf{and}(\mathsf{tt}, \mathsf{ff}) \to \mathsf{and}(q_1(\mathsf{tt}), \mathsf{ff}) \to \mathsf{and}(q_1(\mathsf{tt}), q_0(\mathsf{ff})) \to q_0(\mathsf{and}(\mathsf{tt}, \mathsf{ff}))$$

The result of this derivation is the final state q_0; the term is rejected.

The following picture illustrates a run of the automaton on the bigger term $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{ff}), \mathsf{and}(\mathsf{tt}, \mathsf{ff})))$:



For the sake of conciseness, we applied rules in parallel where possible. At first we apply the rules to the leaves of the term, performing three rewrite steps in parallel. This effectively produces the initial states of the run. Subsequent rule applications propagate the states according to the rules until we obtain the final state at the root of the term.

Note that in both runs, apart from the final state at the root, the result term is the same as the one we started with. This is expected. The only significant output of a DUTA run is the final state.

The rules of a DUTA contain some syntactic overhead as they explicitly copy the function symbol from the left-hand side to the right-hand side. This formulation serves two purposes: first, it makes it possible to describe the run of a DUTA as a term reduction as in the above example. Secondly, we will see that the more sophisticated automata that we will consider later are simply generalisations of the rules of a DUTA, which for example do not require copying the function symbol but allow arbitrary transformations.

2.2 Algebras and Catamorphisms

For the representation of recursion schemes in Haskell, we consider data types as fixed points of polynomial functors:

$$\textbf{data}\ \mathit{Term}\ f = \mathit{In}\ (f\ (\mathit{Term}\ f))$$

Given a functor f that represents some signature, $\mathit{Term}\ f$ constructs its fixed point, which represents the terms over f. For example, the data type *Exp* from the introduction may be instead defined as $\mathit{Term}\ \mathit{Sig}$ with[2]

$$\textbf{data}\ \mathit{Sig}\ e = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ e\ e$$

The functoriality of *Sig* is given by an instance of the type class *Functor*:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ \mathit{Sig}\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{Val}\ i) \quad\ = \mathit{Val}\ i\\ &\quad \mathit{fmap}\ f\ (\mathit{Plus}\ x\ y) = \mathit{Plus}\ (f\ x)\ (f\ y)\end{aligned}$$

The function *eval* from the introduction is defined by a simple recursion scheme: its recursive definition closely follows the recursive definition of the data type *Exp*. This recursion scheme is known as *catamorphism* (or also *fold*). Given an *algebra*, i.e. a functor f and type a together with a function of type $f\ a \to a$, its catamorphism is a function of type $\mathit{Term}\ f \to a$ constructed as follows:

$$\begin{aligned}&\mathit{cata} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to (\mathit{Term}\ f \to a)\\ &\mathit{cata}\ \phi\ (\mathit{In}\ t) = \phi\ (\mathit{fmap}\ (\mathit{cata}\ \phi)\ t)\end{aligned}$$

In the definition of the algebra for the evaluation function, we make use of the fact that the arguments of the *Plus* constructor are already the results of evaluating the corresponding subexpressions:

$$\begin{aligned}&\mathit{evalAlg} :: \mathit{Sig}\ \mathit{Int} \to \mathit{Int}\\ &\mathit{evalAlg}\ (\mathit{Val}\ i) \quad\ = i\\ &\mathit{evalAlg}\ (\mathit{Plus}\ x\ y) = x + y\end{aligned}$$

$$\begin{aligned}&\mathit{eval} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Int}\\ &\mathit{eval} = \mathit{cata}\ \mathit{evalAlg}\end{aligned}$$

Programming in algebras and catamorphisms or other algebraic or coalgebraic recursion schemes is a well-known technique in functional programming [20]. We shall use this representation in order to implement the recursion schemes that we derive from the tree automata.

[2] $\mathit{Term}\ \mathit{Sig}$ is "almost" isomorphic to *Exp*. The only difference stems from the fact that the constructor *In* is non-strict.

2.3 Bottom-Up State Transition Functions

If we omit the syntactic overhead of the state transition rules of DUTAs, we see that DUTAs are algebras – in fact, they were originally defined as such [5]. For instance, the algebra of the automaton in Example 1 is an algebra that evaluates Boolean expressions. Speaking in Haskell terms, a DUTA over a signature functor F is given by a type of states Q, a state transition function in the form of an F-algebra $trans :: F\ Q \to Q$, and a predicate $acc :: Q \to Bool$. A term over F is an element of type $Term\ F$. When running a DUTA on a term t of type $Term\ F$, we obtain the final state $cata\ trans\ t$ of the run. Afterwards, the predicate acc checks whether the final state is accepting:

```
runDUTA :: Functor f ⇒ (f q → q) → (q → Bool) → Term f → Bool
runDUTA trans acc = acc . cata trans
```

Example 2. We implement the DUTA from Example 1 in Haskell as follows:

```
data F a = And a a
         | Not a
         | TT | FF
data Q = Q0 | Q1

acc :: Q → Bool
acc Q1 = True
acc Q0 = False

trans :: F Q → Q
trans FF          = Q0
trans TT          = Q1
trans (Not Q0)    = Q1
trans (Not Q1)    = Q0
trans (And Q1 Q1) = Q1
trans (And _  _ ) = Q0
```

The automaton is run on a term of type $Term\ F$ as follows:

```
evalBool :: Term F → Bool
evalBool = runDUTA trans acc
```

The restriction to a finite state space is not crucial for our purposes as we are not interested in deciding properties of automata. Instead, we want to use automata as powerful recursion schemes that allow for modular definitions of functions on terms. Since we are only interested in the traversal of the term that an automaton provides, we also drop the predicate and consider the final state as the output of a run of the automaton. We, therefore, consider only the transition function of a DUTA:

```
type UpState f q = f q → q
runUpState :: Functor f ⇒ UpState f q → Term f → q
runUpState = cata
```

With the functions $evalAlg$ from Section 2.2 and $trans$ from Example 2, we have already seen two simple examples of bottom-up state transition functions. In practice, only few state transitions of interest are that simple, of course.

In the following, we want to write a simple compiler for our expression language that generates code for a simple virtual machine with a single accumulator

register and a random access memory indexed by non-negative integers. At first, we devise the instructions of the virtual machine:

```
type Addr = Int
data Instr = Acc Int | Load Addr | Store Addr | Add Addr
type Code = [Instr]
```

For simplicity, we use integers to represent addresses for the random access memory. The four instructions listed above write an integer constant to the accumulator, load the contents of a memory cell into the accumulator, store the contents of the accumulator into a memory cell, and add the contents of a memory cell to the contents of the accumulator, respectively.

The code that we want to produce for an expression e of type *Term Sig* should evaluate e, i.e. after executing the code, the virtual machine's accumulator is supposed to contain the integer value *eval e*:

```
codeSt :: UpState Sig Code
codeSt (Val i)      = [Acc i]
codeSt (Plus x y) = x ++ [Store a] ++ y ++ [Add a]
  where a = ...
```

In order to perform addition, the result of the computation for the first summand has to be stored into a temporary memory cell at some address a. However, we also have to make sure that this memory cell is not overwritten by the computation for the second summand. To this end, we maintain a counter that tells us which address is safe to use:

```
codeAddrSt :: UpState Sig (Code, Addr)
codeAddrSt (Val i)                = ([Acc i], 0)
codeAddrSt (Plus (x, a') (y, a)) = (x ++ [Store a] ++ y ++ [Add a],
                                    1 + max a a')
code :: Term Sig → Code
code = fst . runUpState codeAddrSt
```

While this definition yields the desired code generator, it is not very elegant as it mixes the desired output state – the code – with an auxiliary state – the fresh address. This flaw can be mitigated by using a state monad to carry around the auxiliary state. In this way we can still benefit from computing both states side by side has, which means that the input term is only traversed once.

This however still leaves the specification of two computations uncomfortably entangled, which is not only more prone to errors but also inhibits reuse and flexibility: the second component of the state, which we use as a fresh address, is in fact the height of the expression and might be useful for other computations:

```
heightSt :: UpState Sig Int
heightSt (Val _)     = 0
heightSt (Plus x y) = 1 + max x y
```

Moreover, as we extend the expression language with new language features, we might have to change the way we allocate memory locations for intermediate results. Thus, separating the two components of the computation is highly desirable since it would then allow us to replace the *heightSt* component with a different one while reusing the rest of the code generator.

The next section addresses this concern.

3 Making Tree Automata Modular

Our goal is to devise modular recursion schemes. In this section, we show how to leverage two dimensions of modularity inherent in tree automata, viz. the state space and the signature. For each dimension, we present a well-know technique to make use of the modularity in the specification of automata. In particular, we shall demonstrate these techniques on bottom-up state transitions. However, due to their generality, both techniques are applicable also to the more advanced tree automata that we consider in later sections.

3.1 Product Automata

A common construction in automata theory combines two automata by simply forming the cartesian product of their state spaces and defining the state transition componentwise according to the state transitions of the original automata. The resulting automaton runs the original automata in parallel. We shall follow the same idea to construct the state transition *codeAddrSt* from Section 2.3 by combining the state transition *heightSt* with a state transition that computes the machine code using the state maintained by *heightSt*.

However, in contrast to the standard product automaton construction, the two computations in our example are not independent from each other – the code generator depends on the height in order to allocate memory addresses. Therefore, we need a means of communication between the constituent automata.

In order to allow access to components of a compound state space, we define a binary type class $\in$ that tells us if a type is a component of a product type and provides a projection for that component:

$$\begin{array}{l}\textbf{class } a \in b \textbf{ where}\\ \quad pr :: b \to a\end{array}$$

Using *overlapping instance declarations*, we define the relation $a \in b$ as follows:

$$\begin{array}{lll}\textbf{instance} & a \in a & \textbf{where } pr = id\\ \textbf{instance} & a \in (a, b) & \textbf{where } pr = fst\\ \textbf{instance } (c \in b) \Rightarrow & c \in (a, b) & \textbf{where } pr = pr \,.\, snd\end{array}$$

That is, we have $a \in b$ if b is of the form $(b_1, (b_2, ...))$ and $a = b_i$ for some i.

We generalise bottom-up state transitions by allowing the successor state of a transition to be dependent on a potentially larger state space:

$$\textbf{type}\ \mathit{DUpState}\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$$

The result state of type q for the state transition of the above type may depend on the states that are propagated from below. However, in contrast to ordinary bottom-up state transitions, these states – of type p – may contain more components in addition to the component of type q.

Every ordinary bottom-up state transition such as *heightSt* can be readily converted into such a *dependent bottom-up state transition function* by precomposing the projection *pr*:

$$\begin{aligned}&\mathit{dUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{UpState}\ f\ q \to \mathit{DUpState}\ f\ p\ q\\&\mathit{dUpState}\ \mathit{st} = \mathit{st}\ .\ \mathit{fmap}\ \mathit{pr}\end{aligned}$$

A dependent state transition function is the same as an ordinary state transition function if the state spaces p and q coincide. Hence, we can run such a dependent state transition function in the same way:

$$\begin{aligned}&\mathit{runDUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{DUpState}\ f\ q\ q \to \mathit{Term}\ f \to q\\&\mathit{runDUpState}\ f = \mathit{runUpState}\ f\end{aligned}$$

When defining a dependent state transition function, we can make use of the fact that the state propagated from below may contain additional components. For the definition of the state transition function generating the code, we declare that we expect an additional state component of type *Int*.

$$\begin{aligned}&\mathit{codeSt} :: (\mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ \mathit{Code}\\&\mathit{codeSt}\ (\mathit{Val}\ i) \quad\ = [\mathit{Acc}\ i]\\&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = \mathit{pr}\ x +\!\!+ [\mathit{Store}\ a] +\!\!+ \mathit{pr}\ y +\!\!+ [\mathit{Add}\ a]\\&\quad \textbf{where}\ a = \mathit{pr}\ y\end{aligned}$$

Using the method *pr* of the type class $\in$, we project to the desired components of the state: *pr x* and the first occurrence of *pr y* are of type *Code* whereas the second occurrence of *pr y* is of type *Int*.

The product construction that combines two dependent state transition functions is simple: it takes two state transition functions depending on the same (compound) state space and combines them by forming the product of their respective outcomes:

$$\begin{aligned}(\otimes) :: (p \in c, q \in c) \Rightarrow\ & \mathit{DUpState}\ f\ c\ p \to \mathit{DUpState}\ f\ c\ q\\ & \to \mathit{DUpState}\ f\ c\ (p, q)\\(\mathit{sp} \otimes \mathit{sq})\ t = (\mathit{sp}\ t, \mathit{sq}\ t)&\end{aligned}$$

We obtain the desired code generator from Section 2.3 by combining our two (dependent) state transition functions and running the resulting state transition function:

$$\begin{aligned}&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Code}\\&\mathit{code} = \mathit{fst}\ .\ \mathit{runDUpState}\ (\mathit{codeSt} \otimes \mathit{dUpState}\ \mathit{heightSt})\end{aligned}$$

Note that combining state transition functions in this way is not restricted to such simple dependencies. State transition functions may depend on each other. The construction that we have seen in this section makes it possible to decompose state spaces into isolated modules with a typed interface to access them. This practice of decomposing state spaces is not different from the abstraction and reuse that we perform when writing mutual recursive functions. Functions which can be defined in this way are also known as *mutumorphisms* [6].

There are still two minor shortcomings, which we shall address when we consider other types of automata below. First, the extraction of components from compound states is purely based on the type information, which can easily result in confusion of distinct state components that happen to have the same type. This can be seen in the instance declarations for the type class $\in$, which are overlapping and will simply select the left-most occurrence of a type. Secondly, we only allow access to the state of the children of the current node. In principle, this restriction is no problem as we can use the states of the children nodes to compute the state of the current node. For example, if, in the code generation, we needed the height of the current expression instead of the height of the right summand, we could have computed it from the height of both summands. However, this means that code as well as the corresponding computations are duplicated since the state of the current node is already computed by the corresponding state transition.

3.2 Compositional Data Types

We also want to leverage the modularity that stems from the data types on which we want to define functions. This modularity is based on the ability to combine functors by forming coproducts:

```
data (f ⊕ g) e = Inl (f e) | Inr (g e)
instance (Functor f, Functor g) ⇒ Functor (f ⊕ g) where
  fmap f (Inl e) = Inl (fmap f e)
  fmap f (Inr e) = Inr (fmap f e)
```

Using the $\oplus$ operator, we can extend the signature functor *Sig* with an increment operation, for example:

```
data Inc e = Inc e
type Sig'  = Inc ⊕ Sig
```

In order to make use of this composition of functors for defining automata on functors in a modular fashion, we will follow Swierstra's *data types à la carte* [23] approach, which we will summarise briefly below.

The use of coproducts entails that each (sub)term has to be explicitly tagged with zero or more *Inl* or *Inr* tags. In order to add the correct tags automatically, injections are derived using a type class:

```
class sub ≼ sup where
  inj :: sub a → sup a
```

Similarly to the type class $\in$, we define the subsignature relation $\preceq$ as follows:

$$\begin{array}{lll}
\textbf{instance} & f \preceq f & \textbf{where } inj = id \\
\textbf{instance} & f \preceq (f \oplus g) & \textbf{where } inj = Inl \\
\textbf{instance } (f \preceq g) \Rightarrow & f \preceq (h \oplus g) & \textbf{where } inj = Inr \mathbin{.} inj
\end{array}$$

That is, we have $f \preceq g$ if g is of the form $g_1 \oplus (g_2 \oplus ...)$ and $f = g_i$ for some i. From the injection function inj, we derive an injection function for terms:

$$\begin{array}{l}
inject :: (g \preceq f) \Rightarrow g\ (Term\ f) \rightarrow Term\ f \\
inject = In \mathbin{.} inj
\end{array}$$

Additionally, in order to reduce syntactic overhead, we assume, for each signature functor such as Sig or Inc, smart constructors that comprise the injection, e.g.:

$$\begin{array}{l}
plus :: (Sig \preceq f) \Rightarrow Term\ f \rightarrow Term\ f \rightarrow Term\ f \\
plus\ x\ y = inject\ (Plus\ x\ y) \\
inc :: (Inc \preceq f) \Rightarrow Term\ f \rightarrow Term\ f \\
inc\ x = inject\ (Inc\ x)
\end{array}$$

Using these smart constructors, we can write, for example, $inc\ (val\ 3\text{`}plus\text{`}\ val\ 4)$ to denote the expression $inc(3 + 4)$.

For writing modular functions on compositional data types, we use type classes. For example, for recasting the definition of the *heightSt* state transition function, we introduce a new type class and make it propagate over coproducts:

$$\begin{array}{l}
\textbf{class } HeightSt\ f \textbf{ where} \\
\quad heightSt :: UpState\ f\ Int \\
\textbf{instance } (HeightSt\ f, HeightSt\ g) \Rightarrow HeightSt\ (f \oplus g) \textbf{ where} \\
\quad heightSt\ (Inl\ x) = heightSt\ x \\
\quad heightSt\ (Inr\ x) = heightSt\ x
\end{array}$$

The above instance declaration lifts instances of *HeightSt* over coproducts in a straightforward manner. Subsequently, we will omit these instance declarations as they always follow the same pattern and thus can be generated automatically like instances declarations for *Functor*.

We then instantiate this class for each (atomic) signature functor separately:

$$\begin{array}{l}
\textbf{instance } HeightSt\ Sig \textbf{ where} \\
\quad heightSt\ (Val\ _) \quad = 0 \\
\quad heightSt\ (Plus\ x\ y) = 1 + max\ x\ y \\
\textbf{instance } HeightSt\ Inc \textbf{ where} \\
\quad heightSt\ (Inc\ x) \quad = 1 + x
\end{array}$$

Due to the propagation of instances over coproducts, we obtain an instance of *HeightSt* for Sig' for free.

With the help of the type class *HeightSt*, we eventually obtain an extensible definition of the height function.

$$height :: (Functor\ f, HeightSt\ f) \Rightarrow Term\ f \to Int$$
$$height = runUpState\ heightSt$$

Since we have instantiated *HeightSt* for the signature Sig' and all its subsignatures, the function *height* may be given any argument of type $Term\ f$, where f is the Sig' or any of its subsignatures. Moreover, by simply providing further instance declarations for *HeightSt*, we can extend the domain of *height* to further signatures.

4 Bottom-Up Tree Transducers

A compiler usually consists of several stages that perform diverse kinds of transformations on the abstract syntax tree, e.g. renaming variables or removing syntactic sugar. Representing syntax trees as terms, i.e. values of type $Term\ f$, such transformations are functions of type $Term\ f \to Term\ g$ that map terms over some signature to terms over a potentially different signature. Tree transducers are a well-established technique for specifying such transformations [3, 7]. Moreover, there are a number of composition theorems that permit the composition of certain tree transducers such that the transformation function denoted by the composition is equal to the composition of the transformation functions denoted by the original tree transducers [7]. These composition theorems permit us to perform deforestation [26], i.e. eliminating intermediate results by fusing several stages of a compiler to a single tree transducer [16, 25], thus making tree transducers an attractive recursion scheme.

4.1 Deterministic Bottom-Up Tree Transducers

A *deterministic bottom-up tree transducer* ($DUTT$) defines – like a DUTA – for each function symbol a successor state. But, additionally, it also defines an expression that should replace the original function symbol. More formally, a DUTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a set of states Q and a set of transduction rules of the form

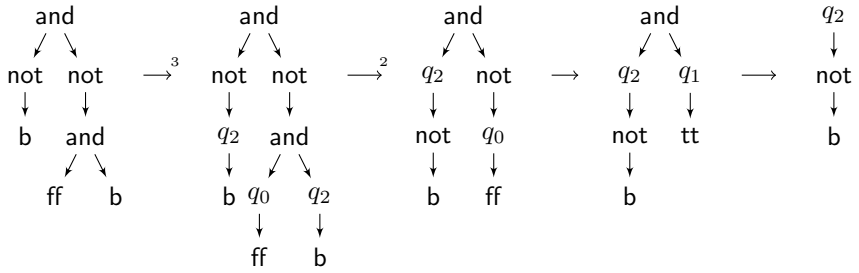
$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(u), \quad \text{with } f \in \mathcal{F} \text{ and } q, q_1, \ldots, q_n \in Q$$

where $u \in \mathcal{T}(\mathcal{G}, \mathcal{X})$ is a term over signature $\mathcal{G}$ and the set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$. Compare this to the state transition rules of DUTAs, which are simply a restriction of the transduction rules above with $u = f(x_1, \ldots, x_n)$, thus only allowing the identity transformation. By repeatedly applying its transduction rules in a bottom-up fashion, a run of a DUTT transforms an input term over $\mathcal{F}$ into an output term over $\mathcal{G}$ plus – similarly to DUTAs – a final state at the root.

Example 3. Consider the signature $\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{ff}/0, \mathsf{tt}/0, \mathsf{b}/0\}$ and the DUTT from $\mathcal{F}$ to $\mathcal{F}$ with $Q = \{q_0, q_1, q_2\}$ and the following transduction rules:

$$\begin{array}{lll}
\mathsf{tt} \to q_1(\mathsf{tt}) & \mathsf{not}(q_0(x)) \to q_1(\mathsf{tt}) & \mathsf{and}(q_1(x), q_1(y)) \to q_1(\mathsf{tt}) \\
\mathsf{ff} \to q_0(\mathsf{ff}) & \mathsf{not}(q_1(x)) \to q_0(\mathsf{ff}) & \mathsf{and}(q_1(x), q_2(y)) \to q_2(y) \\
\mathsf{b} \to q_2(\mathsf{b}) & \mathsf{not}(q_2(x)) \to q_2(\mathsf{not}(x)) & \mathsf{and}(q_2(x), q_1(y)) \to q_2(x) \\
\mathsf{and}(q(x), p(y)) \to q_0(\mathsf{ff}) \quad \text{if } q_0 \in \{p, q\} & & \mathsf{and}(q_2(x), q_2(y)) \to q_2(\mathsf{and}(x, y))
\end{array}$$

The signature $\mathcal{F}$ allows us to express Boolean expression containing a single Boolean variable b. When applied to such an expression, the automaton performs constant folding, i.e. it evaluates subexpression if possible. With the states q_0 and q_1 it signals that a subexpression is false respectively true; q_2 indicates uncertainty. For example, applying the automaton to the expression $\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{not}(\mathsf{and}(\mathsf{ff}, \mathsf{b})))$ yields the following derivation:



The rules for the constant symbols do not perform any transformation in this example and simply provide initial states. Then the first real transformation is performed, which collapses the subterm rooted in and to $q_0(\mathsf{ff})$. The run of the automaton is completed as soon as a state appears at the root, the final state of the run.

4.2 Contexts in Haskell

In order to, represent transduction rules in Haskell, we need a representation of the set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of terms over signature $\mathcal{F}$ and variables $\mathcal{X}$. We call such extended terms *contexts*. These contexts appear on the right-hand side of transduction rules of DUTTs. We obtain a representation of contexts by simply extending the definition of the data type *Term* by an additional constructor:

```
data Context f a = In (f (Context f a)) | Hole a
```

We call this additional constructor *Hole* as we will use it also for things other than variables. For example, the holes in a context may be filled by other contexts over the same signature. The following function substitutes the contexts in the holes into the surrounding context.

```
appCxt :: Functor f ⇒ Context f (Context f a) → Context f a
appCxt (Hole x) = x
appCxt (In t)   = In (fmap appCxt t)
```

Context f is in fact the *free monad* of the functor f with *Hole* and *appCxt* as unit and multiplication operation, respectively. The functoriality of *Context f* is given as follows:

```
instance Functor f ⇒ Functor (Context f) where
  fmap f (Hole v) = Hole (f v)
  fmap f (In t)   = In (fmap (fmap f) t)
```

Recall that the set of terms $\mathcal{T}(\mathcal{F})$ is defined as the set $\mathcal{T}(\mathcal{F}, \emptyset)$ of terms without variables. We can do the same in the Haskell representation and replace our definition of the type *Term* with the following:

```
data Empty
type Term f = Context f Empty
```

Here, *Empty* is simply an empty type.[3] This definition of *Term* allows us to use terms and context in a uniform manner. For example, the function *appCxt* defined above can also be given the type $Context\ f\ (Term\ f) \to Term\ f$. Moreover, this encoding allows us to give a more general type for the injection function:

```
inject :: (g ≼ f) ⇒ g (Context f a) → Context f a
```

The definition of *inject* remains the same. The same also applies to smart constructors; for example, the smart constructor *plus* has now the more general type

```
plus :: (Sig ≼ f) ⇒ Context f a → Context f a → Context f a
```

Most of the time we are using very simple contexts that only consist of a single functor application as constructed by the following function:

```
simpCxt :: Functor f ⇒ f a → Context f a
simpCxt t = In (fmap Hole t)
```

4.3 Bottom-Up Transduction Functions

The transduction rules of a DUTT use placeholder variables x_1, x_2, etc. in order to refer to arguments of function symbols. These placeholder variables can then be used on the right-hand side of a transduction rule. This mechanism makes it possible to rearrange, remove and duplicate the terms that are matched against these placeholder variables. On the other hand, it is not possible to inspect them. For instance, in Example 3, $\mathsf{not}(q_0(\mathsf{ff})) \to q_1(\mathsf{tt})$ would not be a valid transduction rule as we are not allowed to pattern match on the arguments of not. We can only observe the state.

[3] Note that in Haskell, every data type – including *Empty* – is inhabited by $\bot$. Thus the definition of *Term* is not entirely accurate. However, for the sake of simplicity, we prefer this definition over a more precise one such as in [1].

When representing transduction rules as Haskell functions, we have to be careful in order to maintain this restriction on DUTTs. In their categorical representation, Hasuo et al. [11] recognised that the restriction due to placeholder variables in the transduction rules can be enforced by a *naturality* condition. Naturality, in turn, can be represented in Haskell's type system as *parametric polymorphism*. Following this approach, we represent DUTTs from signature functor f to signature functor g with state space q by the following type:

$$\textbf{type}\ \mathit{UpTrans}\ f\ q\ g = \forall\ a\ .\ f\ (q, a) \to (q, \mathit{Context}\ g\ a)$$

In the definition of tree automata, states are used syntactically as a unary function symbol – an argument with state q is written as $q(x)$ in the left-hand side. In the Haskell representation, we use pairs and simply write (q, x).

In the type *UpTrans*, the type variable a represents the type of the placeholder variables. The universal quantification over a makes sure that placeholders can only be used if they appear on the left-hand side and that they cannot be inspected.

Example 4. We implement the DUTT from Example 3 in Haskell. At first we define the signature and the state space.

$$\begin{aligned}&\textbf{data}\ F\ a = \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF} \mid B\\ &\textbf{data}\ Q\ \ \ = \mathit{Q0} \mid \mathit{Q1} \mid \mathit{Q2}\end{aligned}$$

For the definition of the transduction function, we use the smart constructors *and*, *not*, *tt*, *ff* and *b* for the constructors of the signature F. These smart constructors are defined as before, e.g.

$$\begin{aligned}&\mathit{and} :: (F \preceq f) \Rightarrow \mathit{Context}\ f\ a \to \mathit{Context}\ f\ a \to \mathit{Context}\ f\ a\\ &\mathit{and}\ x\ y = \mathit{inject}\ (\mathit{And}\ x\ y)\end{aligned}$$

The definition of the transduction function is a one-to-one translation of the transduction rules of the DUTT from Example 3.

$$\begin{aligned}&\mathit{trans} :: \mathit{UpTrans}\ F\ Q\ F\\ &\mathit{trans}\ \mathit{TT} = (\mathit{Q1}, \mathit{tt}); \qquad \mathit{trans}\ (\mathit{Not}\ (\mathit{Q0}, x)) = (\mathit{Q1}, \mathit{tt})\\ &\mathit{trans}\ \mathit{FF} = (\mathit{Q0}, \mathit{ff}); \qquad \mathit{trans}\ (\mathit{Not}\ (\mathit{Q1}, x)) = (\mathit{Q0}, \mathit{ff})\\ &\mathit{trans}\ B\ \ \ = (\mathit{Q2}, b); \qquad \mathit{trans}\ (\mathit{Not}\ (\mathit{Q2}, x)) = (\mathit{Q2}, \mathit{not}\ (\mathit{Hole}\ x))\\ &\mathit{trans}\ (\mathit{And}\ (q, x)\ (p, y))\\ &\quad \mid q \equiv \mathit{Q0} \vee p \equiv \mathit{Q0} \qquad = (\mathit{Q0}, \mathit{ff})\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q1}, x)\ (\mathit{Q1}, y)) = (\mathit{Q1}, \mathit{tt})\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q1}, x)\ (\mathit{Q2}, y)) = (\mathit{Q2}, \mathit{Hole}\ y)\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q2}, x)\ (\mathit{Q1}, y)) = (\mathit{Q2}, \mathit{Hole}\ x)\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q2}, x)\ (\mathit{Q2}, y)) = (\mathit{Q2}, \mathit{and}\ (\mathit{Hole}\ x)\ (\mathit{Hole}\ y))\end{aligned}$$

Since we do not constrain ourselves to finite state spaces, DUTTs do not add any expressive power to the state transition functions of DUTAs. Each DUTT

can be transformed into an algebra whose catamorphism is the transformation denoted by the DUTT:

```
runUpTrans :: (Functor f, Functor g) ⇒ UpTrans f q g
                                     → Term f → (q, Term g)
runUpTrans trans = cata (appCxt′ . trans)
  where appCxt′ (x, y) = (x, appCxt y)
```

For instance, we run the DUTT from Example 4 as follows:

```
foldBool :: Term F → (Q, Term F)
foldBool = runUpTrans trans
```

As we have seen in Section 3.1, a tree acceptor with a compound state space comprises several computations which may be disentangled in order to increase modularity. A tree transducer intrinsically combines two computations: the state transition and the actual transformation of the term. We will see in Section 4.5 how to disentangle these two components. Before that, we shall look at a special case of DUTTs.

4.4 Tree Homomorphisms

To simplify matters, Bahr and Hvitved [1] focused on tree transducers with a singleton state space, also known as *tree homomorphisms* [3]:

```
type Hom f g = ∀ a . f a → Context g a
runHom :: (Functor f, Functor g) ⇒ Hom f g → Term f → Term g
runHom hom = cata (appCxt . hom)
```

Tree homomorphisms can only transform the tree structure uniformly without the ability to maintain a state. Nonetheless, tree homomorphisms provide a useful recursion scheme. For example, desugaring, i.e. transforming syntactic sugar of a language to the language's core operations, can in many cases be implemented as a tree homomorphism. Reconsider the signature $Sig' = Inc \oplus Sig$ that extends *Sig* with an increment operator. The increment operator is only syntactic sugar for adding the value 1. The corresponding desugaring transformation can be implemented as a tree homomorphism:

```
class DesugHom f g where
  desugHom :: Hom f g
  -- instance declaration lifting DesugHom to coproducts omitted
desugar :: (Functor f, Functor g, DesugHom f g) ⇒ Term f → Term g
desugar = runHom desugHom
instance (Sig ≼ g) ⇒ DesugHom Inc g where
  desugHom (Inc x) = Hole x `plus` val 1
instance (Functor g, f ≼ g) ⇒ DesugHom f g where
  desugHom = simpCxt . inj
```

The first instance declaration states that as long as the target signature g contains Sig, we can desugar the signature Inc to g by mapping $inc(x)$ to $x+1$. Using overlapping instances, the second instance declaration then defines the desugaring for all other signatures f – provided f is contained in the target signature – by leaving the input untouched.

The above instance declarations make it now possible to use the *desugar* function with type $Term\ Sig' \to Term\ Sig$. That is, *desugar* transforms a term over signature Sig' to a term over signature Sig.

As an ordinary recursive Haskell function we would implement desugaring as follows:

```
data Exp  = Val  Int | Plus  Exp  Exp
data Exp' = Val' Int | Plus' Exp' Exp' | Inc' Exp'
desugExp :: Exp' → Exp
desugExp (Val' i)     = Val i
desugExp (Plus' e f) = desugExp e `Plus` desugExp f
desugExp (Inc' e)     = desugExp e `Plus` Val 1
```

Note that we have to provide two separate data types for the input and output types of the function instead of using the compositionality of signatures. Moreover, the function *desugar* is applicable more broadly. It can be used as a function of type $Term\ (f \oplus Inc) \to Term\ f$ for any signature f that contains Sig, i.e. for which we have $Sig \preceq f$. Apart from these advantages in modularity and extensibility we also obtain all the advantages of using a transducer, which we shall discuss in more detail in Section 7.

4.5 Combining Tree Homomorphisms with State Transitions

We aim to combine the simplicity of tree homomorphisms and the expressivity of bottom-up tree transducers. To this end, we shall devise a method to combine a tree homomorphism and a state transition function to form a DUTT. This construction will be complete in the sense that any DUTT can be constructed in this way.

At first, compare the types of automata that we have considered so far:

```
type Hom     f   g = ∀ a . f     a  →     Context g a
type UpState f q   =       f   q    →  q
type UpTrans f q g = ∀ a . f (q, a) → (q, Context g a)
```

We can observe from this – admittedly suggestive – comparison that a bottom-up tree transducer is roughly a combination of a tree homomorphism and a state transition function. Our aim is to make use of this observation by decomposing the specification of a bottom-up tree transducer into a tree homomorphism and a bottom-up state transition function. Like for the product construction of state transition functions from Section 3.1, we have to provide a mechanism to deal with dependencies between the two components. Since the state transition is

independent from the tree transformation, we only need to allow the tree homomorphism to access the state information that is produced by the bottom-up state transition.

A *stateful tree homomorphism* can thus be (tentatively) defined as follows:

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ f\ (q, a) \to \mathit{Context}\ g\ a$$

Since q appears to the left of the function arrow but not to the right, functions of the above type have access to the states of the arguments, but do not transform the state themselves. However, we want to make it easy to ignore the state if it is not needed as the state is often only needed for a small number of cases. This goal can be achieved by replacing the pairing with the state space q by an additional argument of type $a \to q$.

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ (a \to q) \to f\ a \to \mathit{Context}\ g\ a$$

We can still push this interface even more to the original tree homomorphism type *Hom* by turning the function argument into an implicit parameter [18]:

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ (?\mathit{state} :: a \to q) \Rightarrow f\ a \to \mathit{Context}\ g\ a$$

In a last refinement step, we add an implicit parameter that provides access to the state of the current node as well:

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ (?\mathit{above} :: q, ?\mathit{below} :: a \to q) \Rightarrow f\ a \to \mathit{Context}\ g\ a$$

Functions with implicit parameters have to be invoked in the scope of appropriate bindings. For functions of the above type this means that $?\mathit{below}$ has to be bound to a function of type $a \to q$ and $?\mathit{above}$ to a value of type q. We shall use the following function to make implicit parameters explicit:

$$\begin{aligned}&\mathit{explicit} :: ((?\mathit{above} :: q, ?\mathit{below} :: a \to q) \Rightarrow b) \to q \to (a \to q) \to b\\ &\mathit{explicit}\ x\ ab\ be = x\ \textbf{where}\ ?\ \mathit{above} = ab; ?\mathit{below} = be\end{aligned}$$

In particular, given a stateful tree homomorphism h of type $\mathit{QHom}\ f\ q\ g$, we thus obtain a function $\mathit{explicit}\ h$ of type $q \to (a \to q) \to f\ a \to \mathit{Context}\ g\ a$.

The use of implicit parameters is solely for reasons of syntactic appearance and convenience. One can think of implicit parameters as reader monads without the syntactic overhead of monads. If, in the definition of a stateful tree homomorphism, the state is not needed, it can be easily ignored. Hence, tree homomorphisms are, in fact, also syntactic special cases of stateful tree homomorphisms.

The following construction combines a stateful tree homomorphism of type $\mathit{QHom}\ f\ q\ g$ and a state transition function of type $\mathit{UpState}\ f\ q$ into a tree transducer of type $\mathit{UpTrans}\ f\ q\ g$, which can then be used to perform the desired transformation:

$$\begin{aligned}&\mathit{upTrans} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow\\ &\qquad \mathit{UpState}\ f\ q \to \mathit{QHom}\ f\ q\ g \to \mathit{UpTrans}\ f\ q\ g\end{aligned}$$

```
upTrans st hom t = (q, c) where
  q = st (fmap fst t)
  c = fmap snd (explicit hom q fst t)
runUpHom :: (Functor f, Functor g) ⇒
                UpState f q → QHom f q g → Term f → (q, Term g)
runUpHom st hom = runUpTrans (upTrans st hom)
```

Often the state space accessed by a stateful tree homomorphism is compound. Therefore, it is convenient to have the projection function *pr* built into the interface to the state space:

```
above :: (?above :: q, p ∈ q) ⇒ p
above = pr ? above
below :: (?below :: a → q, p ∈ q) ⇒ a → p
below = pr . ?below
```

In order to illustrate how stateful tree homomorphisms are programmed, we extend the signature *Sig* with variables and let bindings:

```
type Name = String
data Let e  = LetIn Name e e | Var Name
type LetSig = Let ⊕ Sig
```

We shall implement a simple optimisation that removes let bindings whenever the variable that is bound is not used in the scope of the let binding. To this end, we define a state transition that computes the set of free variables:

```
type Vars = Set Name
class FreeVarsSt f where
  freeVarsSt :: UpState f Vars
instance FreeVarsSt Sig where
  freeVarsSt (Plus x y) = x `union` y
  freeVarsSt (Val _)    = empty
instance FreeVarsSt Let where
  freeVarsSt (Var v)       = singleton v
  freeVarsSt (LetIn v e s) = if v `member` s then delete v (e `union` s)
                                             else s
```

Note that the free variables occurring in the right-hand side of a binding are only included if the bound variable occurs in the scope of the let binding. The transformation itself is simple:

```
class RemLetHom f q g where
  remLetHom :: QHom f q g
instance (Vars ∈ q, Let ≼ g, Functor g) ⇒ RemLetHom Let q g where
  remLetHom (LetIn v _ s) | ¬ (v `member` below s) = Hole s
```

```
  remLetHom t                                          = simpCxt (inj t)
instance (Functor f, Functor g, f ≼ g) ⇒ RemLetHom f q g where
  remLetHom = simpCxt . inj
```

The homomorphism removes a let binding whenever the bound variable is not found in the set of free variables. Otherwise, no transformation is performed. Notice that the type specifies that the transformation depends on a state space that at least contains a set of variables. In addition, we make use of overlapping instances to define the transformation for all signatures different from *Let*. We then obtain the desired transformation function by combining the stateful tree homomorphism with the state transition computing the free variables:

```
remLet :: (Functor f, FreeVarsSt f, RemLetHom f Vars f)
       ⇒ Term f → Term f
remLet = snd . runUpHom freeVarsSt remLetHom
```

In particular, we can give *remLet* the type $Term\ LetSig \to Term\ LetSig$ but also $Term\ (Inc \oplus LetSig) \to Term\ (Inc \oplus LetSig)$.

4.6 Refining Dependent Bottom-Up State Transition Functions

The implicit parameters ?*below* and ?*above* of stateful tree homomorphisms provide an interface to the states of the children of the current node as well as the state of the current node itself. The same interface can be given to dependent bottom-up state transition functions as well. We therefore redefine the type of these state transitions from Section 3.1 as follows:

```
type DUpState f p q = ∀ a . (?below :: a → p, ?above :: p, q ∈ p) ⇒ f a → q
```

While the definition of the product operator $\otimes$ remains the same, we have to change the other functions slightly to accommodate this change:

```
dUpState :: Functor f ⇒ UpState f q → DUpState f p q
dUpState st = st . fmap below

upState :: DUpState f q q → UpState f q
upState st s = res where
  res = explicit st res id s

runDUpState :: Functor f ⇒ DUpState f q q → Term f → q
runDUpState = runUpState . upState
```

Note that definition of *res* in *upState* is cyclic and thus crucially depends on Haskell's non-strict semantics. This also means that dependent state transition functions do not necessarily yield a terminating run since one can create a cyclic dependency by defining a state transition that depends on its own result such as the following:

```
loopSt :: DUpState f p q
loopSt _ = above
```

The definition of the code generator from Section 3.1 is easily adjusted to the slightly altered interface of dependent state transitions. Since we intend to extend the code generator in Section 6, we also turn it into a type class:

```
class CodeSt f q where
  codeSt :: DUpState f q Code
code :: (Functor f, CodeSt f (Code, Int), HeightSt f)
     ⇒ Term f → (Code, Addr)
code = runDUpState (codeSt ⊗ dUpState heightSt)
instance (Int ∈ q) ⇒ CodeSt Sig q where
  codeSt (Val i)    = [Acc i]
  codeSt (Plus x y) = below x ++ [Store a] ++ below y ++ [Add a]
    where a = below y
```

Note that the access to the state of the current node – via *above* – solves one of the minor issues we have identified at the end of Section 3.1. In order to obtain the state of the current node, we do not have to duplicate the corresponding state transition anymore. Moreover, we can use the same interface when we move to top-down state transitions in the next section.

5 Top-Down Automata

Operations on abstract syntax trees are often dependent on a state that is propagated top-down rather than bottom-up, e.g. typing environments and variable bindings. For such operations, recursion schemes derived from bottom-up automata are not sufficient. Hence, we shall consider top-down automata as a complementary paradigm to overcome this restriction.

Unlike the bottom-up case, we will not start with acceptors but with transducers. Our interest for bottom-up acceptors was based on the fact that such automata produce an output state. For top-down acceptors this application vanishes since such automata rather consume an input state than produce an output state. We will however come back to top-down state transition in order to make the state transition of top-down transducer modular – using the same stateful tree homomorphisms that we introduced in Section 4.5.

5.1 Deterministic Top-Down Tree Transducers

Deterministic top-down tree transducers (DDTTs) are able to produce transformations that depend on a top-down flow of information. They work in a fashion similar to bottom-up tree transducers but propagate their state downwards rather than upwards. More formally, a DDTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a set of states Q, an initial state $q_0 \in Q$ and a set of transduction rules of the form

$$q(f(x_1, \ldots, x_n)) \to u \quad \text{with } f \in \mathcal{F} \text{ and } q \in Q$$

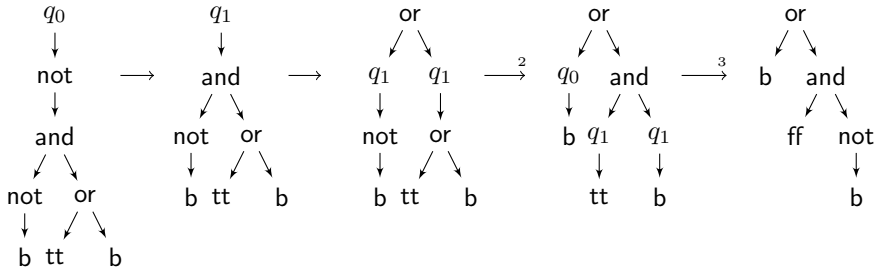
where $u \in \mathcal{T}(\mathcal{G}, Q(\mathcal{X}))$ is a term over $\mathcal{G}$ and $Q(\mathcal{X}) = \{p(x_i) \mid p \in Q, 1 \leq i \leq n\}$. That is, the right-hand side is a term that may have subterms of the form $p(x_i)$ with x_i a variable from the left-hand side and p a state in Q. In other words, each *occurrence* of a variable on the right-hand side is given a successor state.

In order to run a DDTT on a term $t \in \mathcal{T}(\mathcal{F})$, we have to provide an initial state q_0 and then apply the transduction rules to $q_0(t)$ in a top-down fashion. Eventually, this yields a result term $t' \in \mathcal{T}(\mathcal{G})$.

Example 5. Consider the signature $\mathcal{F} = \{\mathsf{or}/2, \mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0, \mathsf{b}/0\}$ and the DDTT from $\mathcal{F}$ to $\mathcal{F}$ with the set of states $Q = \{q_0, q_1\}$, initial state q_0 and the following transduction rules:

$$
\begin{array}{llll}
q_0(\mathsf{b}) \to \mathsf{b} & q_0(\mathsf{tt}) \to \mathsf{tt} & q_0(\mathsf{ff}) \to \mathsf{ff} & q_0(\mathsf{not}(x)) \to q_1(x) \\
q_1(\mathsf{b}) \to \mathsf{not}(\mathsf{b}) & q_1(\mathsf{tt}) \to \mathsf{ff} & q_1(\mathsf{ff}) \to \mathsf{tt} & q_1(\mathsf{not}(x)) \to q_0(x) \\
q_0(\mathsf{and}(x,y)) \to \mathsf{and}(q_0(x), q_0(y)) & & q_0(\mathsf{or}(x,y)) \to \mathsf{or}(q_0(x), q_0(y)) & \\
q_1(\mathsf{and}(x,y)) \to \mathsf{or}(q_1(x), q_1(y)) & & q_1(\mathsf{or}(x,y)) \to \mathsf{and}(q_1(x), q_1(y)) &
\end{array}
$$

Terms over $\mathcal{F}$ are Boolean expressions with a single Boolean variable b. The above DDTT transforms such an expression into negation normal form by moving the operator not inwards. For instance, applied to the Boolean expression $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{or}(\mathsf{tt}, \mathsf{b})))$, the automaton yields the following derivation:



In order to start the run of a DDTT, the initial state q_0 has to be explicitly inserted at the root of the input term. The run of the automaton is completed as soon as all states in the term have vanished; there is no final state.

5.2 Top-Down Transduction Functions

Similar to bottom-up tree transducers, we follow the *placeholders-via-naturality* principle of Hasuo et al. [11] in order to represent top-down transduction functions:

$$\textbf{type}\ \mathit{DownTrans}\ f\ q\ g = \forall\ a\ .\ (q, f\ a) \to \mathit{Context}\ g\ (q, a)$$

Now the state comes from above and is propagated downwards to the holes of the context, which defines the actual transformation that the transducer performs.

Running a top-down tree transducer on a term is a straightforward affair:

```
runDownTrans :: (Functor f, Functor g) ⇒ DownTrans f q g → q
                                       → Term f → Term g
runDownTrans tr q t = run (q, t) where
  run (q, In t) = appCxt (fmap run (tr (q, t)))
```

A top-down transducer is run by applying its transduction function – *tr* (q, t) – then recursively running the transformation in the holes of the produced context – *fmap run* – and finally joining the context with the thus produced embedded terms – *appCxt*.

Example 6. We implement the DDTT from Example 5 in Haskell as follows:

```
data F a = Or a a | And a a | Not a | TT | FF | B
data Q = Q0 | Q1

trans :: DownTrans F Q F
trans (Q0, TT) = tt;          trans (Q0, B) = b
trans (Q1, TT) = ff;          trans (Q1, B) = not b
trans (Q0, FF) = ff;          trans (Q0, Not x) = Hole (Q1, x)
trans (Q1, FF) = tt;          trans (Q1, Not x) = Hole (Q0, x)
trans (Q0, And x y) = Hole (Q0, x) `and` Hole (Q0, y)
trans (Q1, And x y) = Hole (Q1, x) `or`  Hole (Q1, y)
trans (Q0, Or  x y) = Hole (Q0, x) `or`  Hole (Q0, y)
trans (Q1, Or  x y) = Hole (Q1, x) `and` Hole (Q1, y)
```

The definition of the transduction function *trans* is a one-to-one translation of the transduction rules of the DDTT from Example 5. Note, that we use the smart constructors *or*, *and*, *not*, *tt*, *ff* and *b* on the right-hand side of the definitions. We apply the thus defined DDTT to a term of type *Term F* as follows:

```
negNorm :: Term F → Term F
negNorm = runDownTrans trans Q0
```

5.3 Top-Down State Transition Functions

Unfortunately, we cannot provide a full decomposition of DDTTs into a state transition and a homomorphism part in the way we did for DUTTs in Section 4.5. Unlike in DUTTs, the state transition in a DDTT is inherently dependent on the transformation: since a placeholder variable may be copied on the right-hand side, each copy may be given a different successor state! For example, a DDTT may have a transduction rule

$$q_0(f(x)) \to g(q_1(x), q_2(x))$$

which transforms a function symbol f into g and copies the argument of f. However, the two copies are given different successor states, viz. q_1 and q_2.

In order to avoid this dependency of state transitions on the transformation, we restrict ourselves to DDTTs in which successor states are given to placeholder variables and not their occurrences. That is, for each two occurrences of subterms $q_1(x)$ and $q_2(x)$ on the right-hand side of a transduction rule, we require that $q_1 = q_2$. The DDTT given in Example 5 is, in fact, of this form.

The top-down state transitions we are aiming for are dual to bottom-up state transitions. The run of a bottom-up state transition function assigns a state to each node by an upwards state propagation, performing the same computation as an upwards accumulation [8]. The run of a top-down state transition function, on the other hand, should do the same by a downwards state propagation and thus perform the same computation as a downwards accumulation [9, 10].

However, representing top-down state transitions is known to be challenging [8, 9, 10]. A first attempt yields the type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$. This type, however, allows apart from the state transition also a transformation. The result is not required to have the same shape as the input. For example, the following equation (partially) defines a function *bad* of type $\forall\ a\ .\ (Q, \mathit{Sig}\ \ a) \to \mathit{Sig}\ \ Q$:

$$\mathit{bad}\ (q, \mathit{Plus}\ \ x\ \ y) = \mathit{Val}\ 1$$

In order to assign a successor state to each child of the input node without permitting changes to its structure, we use explicit placeholders to which we can assign the successor states:

$$\textbf{type}\ \mathit{DownState}\ f\ \ q = \forall\ i\ .\ \mathit{Ord}\ \ i \Rightarrow (q, f\ \ i) \to \mathit{Map}\ \ i\ \ q$$

The type $\mathit{Map}\ \ i\ \ q$ represents finite mappings from type i to type q. Since such finite mappings are implemented by search trees, we require that the domain type i is of class *Ord*, which provides a total ordering.

The idea is to produce, from a state transition function of the above type, a function of type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$ that does preserve the structure of the input and only produces the successor states. This is achieved by injecting unique placeholders of type i into a value of type $f\ \ a$ – one for each child node. We can then produce the desired value of type $f\ \ q$ from the mapping of type $\mathit{Map}\ \ i\ \ q$ given by the state transition function. A placeholder that is not mapped to a state explicitly is assumed to keep the state of the current node by default.

To work with finite mappings, we assume an interface with $\emptyset$ denoting the empty mapping, $x \mapsto y$ the singleton mapping that maps x to y, $m \cup n$ the left-biased union of two mappings m and n, and a lookup function $\mathit{lookup} :: \mathit{Ord}\ \ i \Rightarrow i \to \mathit{Map}\ \ i\ \ q \to \mathit{Maybe}\ \ q$. Moreover, we define the lookup with default as follows:

$$\begin{aligned}
&\mathit{findWithDefault} :: \mathit{Ord}\ \ i \Rightarrow q \to i \to \mathit{Map}\ \ i\ \ q \to q\\
&\mathit{findWithDefault}\ \ \mathit{def}\ \ i\ \ m = \textbf{case}\ \mathit{lookup}\ \ i\ \ m\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{def}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ \ q\ \ \ \to q
\end{aligned}$$

At first, we need a mechanism to introduce unique placeholders into the structure of a functorial value. To this end, we will use the standard Haskell type class *Traversable* that provides the method

$$mapM :: (Traversable\ f, Monad\ m) \Rightarrow (a \to m\ b) \to f\ a \to m\ (f\ b)$$

which allows us to apply a monadic function to the components of a functorial value and then sequence the resulting monadic effects. Every polynomial functor can be made an instance of *Traversable*. Declarations to that effect can be derived automatically.

Ultimately, we want to number the elements in a functorial value to make them unique placeholders. To this end, we introduce a type of numbered values.

```
newtype Numbered a = Numbered (Int, a)

unNumbered :: Numbered a → a
unNumbered (Numbered (_, x)) = x

instance Eq (Numbered a) where
  Numbered (i, _) ≡ Numbered (j, _) = i ≡ j

instance Ord (Numbered a) where
  compare (Numbered (i, _)) (Numbered (j, _)) = compare i j
```

The instance declarations allow us to use elements of the type *Numbered a* as placeholders.

With the help of the *mapM* combinator, we define a function that numbers the components in a functorial value by counting up using a state monad:

```
number :: Traversable f ⇒ f a → f (Numbered a)
number x = fst (runState (mapM run x) 0) where
  run b = do n ← get
             put (n + 1)
             return (Numbered (n, b))
```

where $runState :: State\ s\ a \to s \to (a, s)$ runs a state monad with state type s, $put :: s \to State\ s\ m\ ()$ sets the state and $get :: State\ s\ s$ queries the state inside a state monad.

Using the above numbering combinator to create unique placeholders, we construct the explicit top-down propagation of states from a mapping of placeholders to successor states. Since the mapping of placeholders to successor states is partial, we also have to give a default state:

```
appMap :: Traversable f ⇒ (∀ i . Ord i ⇒ f i → Map i q)
                        → q → f a → f (q, a)
appMap qmap q s = fmap qfun s' where
  s'       = number s
  qfun k = (findWithDefault q k (qmap s'), unNumbered k)
```

Finally, we can combine a top-down state transition function with a stateful tree homomorphism by propagating the successor states using the *appMap* combinator. As the default state, we take the state of the current node, i.e. by default the state remains unchanged.

$$
\begin{aligned}
&\mathit{downTrans} :: \mathit{Traversable}\ f \Rightarrow \mathit{DownState}\ f\ q \to \mathit{QHom}\ f\ q\ g \\
&\qquad\qquad \to \mathit{DownTrans}\ f\ q\ g \\
&\mathit{downTrans}\ st\ f\ (q, s) = \mathit{explicit}\ f\ q\ \mathit{fst}\ (\mathit{appMap}\ (\mathit{curry}\ st\ q)\ q\ s) \\
&\mathit{runDownHom} :: (\mathit{Traversable}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{DownState}\ f\ q \\
&\qquad\qquad \to \mathit{QHom}\ f\ q\ g \to q \to \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{runDownHom}\ st\ h = \mathit{runDownTrans}\ (\mathit{downTrans}\ st\ h)
\end{aligned}
$$

Note that we use the same type of stateful tree homomorphisms that we introduced for bottom-up state transitions. The roles of ?*above* and ?*below* are simply swapped: ?*above* refers to the state propagated from above whereas ?*below* provides the successor states of the current subterm. Stateful tree homomorphisms are ignorant of the direction in which the state is propagated.

Example 7. We reconstruct the DDTT from Example 6 by defining the state transition and the transformation separately:

$$
\begin{aligned}
&\mathit{state} :: \mathit{DownState}\ F\ Q \\
&\mathit{state}\ (\mathit{Q0}, \mathit{Not}\ x) = x \mapsto \mathit{Q1} \\
&\mathit{state}\ (\mathit{Q1}, \mathit{Not}\ x) = x \mapsto \mathit{Q0} \\
&\mathit{state}\ \_ \qquad\qquad = \emptyset \\
&\mathit{hom} :: \mathit{QHom}\ F\ Q\ F \\
&\mathit{hom}\ \mathit{TT} \qquad = \textbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \textbf{then}\ \mathit{tt}\ \textbf{else}\ \mathit{ff} \\
&\mathit{hom}\ \mathit{FF} \qquad = \textbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \textbf{then}\ \mathit{ff}\ \textbf{else}\ \mathit{tt} \\
&\mathit{hom}\ B \qquad = \textbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \textbf{then}\ b\ \ \textbf{else}\ \mathit{not}\ b \\
&\mathit{hom}\ (\mathit{Not}\ x) \quad = \mathit{Hole}\ x \\
&\mathit{hom}\ (\mathit{And}\ x\ y) = \textbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \textbf{then}\ \mathit{Hole}\ x\ \text{`}\mathit{and}\text{`}\ \mathit{Hole}\ y \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{Hole}\ x\ \text{`}\mathit{or}\text{`}\ \ \mathit{Hole}\ y \\
&\mathit{hom}\ (\mathit{Or}\ x\ y) \quad = \textbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \textbf{then}\ \mathit{Hole}\ x\ \text{`}\mathit{or}\text{`}\ \ \mathit{Hole}\ y \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{Hole}\ x\ \text{`}\mathit{and}\text{`}\ \mathit{Hole}\ y
\end{aligned}
$$

Note that in the definition of the state transition function, we return the empty mapping for all constructors different from *Not*. Consequently, the input state for these constructors is propagated unchanged by default.

By combining the state transition function and the stateful homomorphism, we obtain the same transformation function as in Example 6.

$$
\begin{aligned}
&\mathit{negNorm}' :: \mathit{Term}\ F \to \mathit{Term}\ F \\
&\mathit{negNorm}' = \mathit{runDownHom}\ \mathit{state}\ \mathit{hom}\ \mathit{Q0}
\end{aligned}
$$

Instead of introducing explicit placeholders in order to distribute the successor state, we could have also simply taken the encoding we first suggested, i.e. via a function ρ of type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$, and required as (an unchecked) side condition that ρ must preserve the shape of the input. This approach was taken in Gibbons' generic downwards accumulations [10] in which he requires the accumulation operation to be shape preserving.

Alternatively, we could have also adopted Gibbons' earlier approach to downwards accumulations [9], which instead represents the downward flow of information as a fold over a separately constructed data type called *path*. This path data type is constructed as the fixed point of a functor that is constructed from the signature functor. Unfortunately, this functor is quite intricate and not easy to program with in practice. Apart from that, it would be difficult to construct this path functor for each signature functor in Haskell.

In the end, our approach yields a straightforward representation of downward state transitions that is easy to work with in practise. Moreover, the ability to have a default behaviour for unspecified transitions makes for compact specifications as we have seen in Example 7. However, this default behaviour may also lead to errors more easily due to forgotten transitions.

5.4 Making Top-Down State Transition Functions Modular

Analogously to bottom-up state transition functions, we also define a variant of top-down state transition functions that has access to a bigger state space whose components are defined separately.

$$\begin{aligned}&\textbf{type } \mathit{DDownState}\ f\ p\ q = \forall\ i\ .\ (\mathit{Ord}\ i, ?\mathit{below} :: i \to p, ?\mathit{above} :: p, q \in p)\\ &\qquad\qquad\qquad\qquad \Rightarrow f\ i \to \mathit{Map}\ i\ q\end{aligned}$$

Translations between ordinary top-down state transitions and their generalised variants are produced as follows:

$$\begin{aligned}&\mathit{dDownState} :: \mathit{DownState}\ f\ q \to \mathit{DDownState}\ f\ p\ q\\ &\mathit{dDownState}\ f\ t = f\ (\mathit{above}, t)\\ &\mathit{downState} :: \mathit{DDownState}\ f\ q\ q \to \mathit{DownState}\ f\ q\\ &\mathit{downState}\ f\ (q, s) = \mathit{res}\ \textbf{where}\\ &\quad \mathit{res}\ \ = \mathit{explicit}\ f\ q\ \mathit{bel}\ s\\ &\quad \mathit{bel}\ k = \mathit{findWithDefault}\ q\ k\ \mathit{res}\end{aligned}$$

Similarly to their bottom-up counterparts, dependent top-down state transition functions that depend on the same state space can be combined to form a product state transition:

$$\begin{aligned}&(\circledast) :: (p \in c, q \in c) \Rightarrow \mathit{DDownState}\ f\ c\ p \to \mathit{DDownState}\ f\ c\ q\\ &\qquad\qquad\qquad\qquad \to \mathit{DDownState}\ f\ c\ (p, q)\\ &(\mathit{sp} \circledast \mathit{sq})\ t = \mathit{prodMap}\ \mathit{above}\ \mathit{above}\ (\mathit{sp}\ t)\ (\mathit{sq}\ t)\\ &\mathit{prodMap} :: \mathit{Ord}\ i \Rightarrow p \to q \to \mathit{Map}\ i\ p \to \mathit{Map}\ i\ q \to \mathit{Map}\ i\ (p, q)\end{aligned}$$

This construction is based on the pointwise product of mappings defined by *prodMap*, which we do not give in detail here. Since the mappings are partial, we have to provide a default state that is used in case only one of the mappings has a value for a given index. In accordance with the default behaviour of top-down state transition functions, this default state is the state from above.

As an example, we will define a transformation that replaces variables bound by let expressions with de Bruijn indices. For the sake of demonstration, we will implement this transformation using two states: the scope level, i.e. the number of let-bindings that are in scope, and a mapping from bound variables to the scope level of their respective binding site.

The scope level state simply counts the nesting of let bindings:

```
class ScopeLvlSt f where
  scopeLvlSt :: DownState f Int
instance ScopeLvlSt Let where
  scopeLvlSt (d, LetIn _ _ b) = b ↦ (d + 1)
  scopeLvlSt _                = ∅
instance ScopeLvlSt f where
  scopeLvlSt _                = ∅
```

Here we use the fact that if a successor state is not defined for a subexpression, then the current state is propagated by default.

The state that maintains a mapping from variables to the scope level of their respective binding site is dependent on the scope level state:

```
type VarLvl = Map Name Int
class VarLvlSt f q where
  varLvlSt :: DDownState f q VarLvl
instance (Int ∈ q) ⇒ VarLvlSt Let q where
  varLvlSt (LetIn v _ b) = b ↦ ((v ↦ above) ∪ above)
  varLvlSt _             = ∅
instance VarLvlSt f q where
  varLvlSt _ = ∅
```

Note that the first occurrence of *above* is of type *Int* – derived from the type constraint $Int \in q$ – whereas the second occurrence is of type *VarLvl* – derived from the type constraint $VarLvl \in q$ in the type *DDownState f q VarLvl*.

Since we want to replace explicit variables with de Bruijn indices, we have to replace the signature *Let* with the following signature in the output term:

```
data Let' e   = LetIn' e e | Var' Int
type LetSig' = Let' ⊕ Sig
```

The actual transformation is defined as a stateful tree homomorphism:

```
class DeBruijnHom f q g where
  deBruijnHom :: QHom f q g
instance (VarLvl ∈ q, Int ∈ q, Let' ≼ g) ⇒ DeBruijnHom Let q g where
  deBruijnHom (LetIn _ a b) = letIn' (Hole a) (Hole b)
  deBruijnHom (Var v)       = case lookup v above of
                                Nothing → error "free variable"
```

```
                                    Just i → var' (above − i)
instance (Functor f, Functor g, f ≼ g) ⇒ DeBruijnHom f q g where
  deBruijnHom = simpCxt . inj
```

Note that we issue an error if we encounter a variable that is not bound by a let expression. Otherwise, we create the de Bruijn index by subtracting the variable's scope level from the current scope level.

Finally, we have to tie the components together by forming the product state transition and providing an initial state:

```
deBruijn :: Term LetSig → Term LetSig'
deBruijn = runDownHom stateTrans deBruijnHom init
  where init = (∅, 0) :: (VarLvl, Int)
        stateTrans :: DownState LetSig (VarLvl, Int)
        stateTrans = downState (varLvlSt ⊛ dDownState scopeLvlSt)
```

Due to its open definition, we can give the function *deBruijn* also the type $Term\ (Inc \oplus LetSig) \to Term\ (Inc \oplus LetSig')$, for example.

6 Bidirectional State Transitions

We have seen recursion schemes that use an upwards flow of information as well as recursion schemes that use a downwards flow of information. Some computations, however, require the combination of both. For example, if we want to extend the code generator from Section 4.6 to also work on let bindings, we need to propagate the generated code *upwards* but the symbol table for bound variables *downwards*.

In this section, we show two ways of achieving this combination.

6.1 Avoiding the Problem

The issue of combining two directions of information flow is usually circumvented by splitting up the computation in several runs instead. For the code generator, for instance, we can introduce a preprocessing step that translates let bindings into explicit assignments to memory addresses and variables into corresponding references to memory addresses.

This preprocessing step is easily implemented by modifying the stateful tree homomorphism from Section 5.4 that transforms variables into de Bruijn indices. Instead of de Bruijn indices we generate memory addresses.

At first, we define the signature that contains explicit addresses for bound variables:

```
data LetAddr e = LetAddr Addr e e | VarAddr Addr
type AddrSig   = LetAddr ⊕ Sig
```

The following stateful homomorphism then transforms a term over a signature containing *Let* into a signature containing *LetAddr* instead. The homomorphism depends on the same state as the de Bruijn homomorphism from Section 5.4:

```
class AddrHom f q g where
  addrHom :: QHom f q g
instance (VarLvl ∈ q, Int ∈ q, LetAddr ≼ g) ⇒ AddrHom Let q g where
  addrHom (LetIn _ x y) = letAddr above (Hole x) (Hole y)
  addrHom (Var v)       = case lookup v above of
                            Nothing → error "free variable"
                            Just a → varAddr a
instance (Functor f, Functor g, f ≼ g) ⇒ AddrHom f q g where
  addrHom = simpCxt . inj
```

By combining all components of the computation including the state transition functions *varLvlSt* and *scopeLvlSt* from Section 5.4, we obtain the desired transformation:

```
toAddr :: Addr → Term LetSig → Term AddrSig
toAddr startAddr = runDownHom stateTrans addrHom init
  where init = (∅, startAddr) :: (VarLvl, Int)
        stateTrans :: DownState LetSig (VarLvl, Int)
        stateTrans = downState (varLvlSt ⊛ dDownState scopeLvlSt)
```

The additional argument of type *Addr* allows us to control from which address we should start when assigning addresses to variables.

The actual code generation can then proceed on the signature *LetAddr* instead of *Let*:

```
instance CodeSt LetAddr q where
  codeSt (LetAddr a s e) = below s ++ [Store a] ++ below e
  codeSt (VarAddr a)     = [Load a]
```

To this end, we must also extend the *HeightSt* type class, which is used by the code generator:

```
instance HeightSt LetAddr where
  heightSt (LetAddr _ x y) = 1 + max x y
  heightSt (VarAddr _)     = 0
```

Now, we can use the function *code* from Section 4.6 with the type

```
code :: Term AddrSig → (Code, Addr)
```

Combining this function with the above defined transformation *toAddr*, yields the desired code generator:

```
codeLet :: Term LetSig → Code
codeLet t = c
  where t'          = toAddr (addr + 1) t
        (c, addr)   = code t'
```

When combining the two functions *toAddr* and *code*, we have to be careful to avoid clashes in the use of addresses for storing intermediate results on the one hand and for storing results of let bindings on the other hand. To this end, we use the result *addr* of the code generator function *code*, which is the highest address used for intermediate results, to initialise the address counter for the transformation *toAddr*. This makes sure that we use different addresses for intermediate results and bound variables.

6.2 A Direct Implementation

An alternative approach performs the bottom-up and the top-down computations side-by-side, taking advantage of the non-strict semantics of Haskell. This approach avoids the construction of an intermediate syntax tree that contains the required information.

For implementing a suitable recursion scheme, we make use of the fact that both bottom-up as well as top-down state transition functions in their dependent form share the same interface to access other components of the state space via the implicit parameters ?*above* and ?*below*.

The following combinator runs a bottom-up and a top-down state transition function that both depend on the product of the state spaces they define:

```
runDState :: Traversable f ⇒ DUpState f (u, d) u
                           → DDownState f (u, d) d → d → Term f → u
runDState up down d (In t) = u where
  bel (Numbered (i, s)) =
    let d' = findWithDefault d (Numbered (i, ⊥)) qmap
    in Numbered (i, (runDState up down d' s, d'))
  t'    = fmap bel (number t)
  qmap  = explicit down (u, d) unNumbered t'
  u     = explicit up (u, d) unNumbered t'
```

The definition of *runDState* looks convoluted but follows a simple structure: the two lines at the bottom apply both state transition functions at the current node. To this end, the state from above and the state from below is given as (u, d) and *unNumbered*, respectively. The latter works as t' is computed by first numbering the child nodes and then using the numbering to lookup the successor states from *qmap* as well as recursively applying *runDState* at the child nodes.

Note that the definition of *runDState* is cyclic in several different ways and thus essentially depends on Haskell's non-strict semantics: the result u of the bottom-up state transition function is used also as input for the bottom-up state transition function. Likewise the result *qmap* of the top-down state transition

function is fed into the construction of t', which is given as argument to the top-down state transition function. Moreover, the definition of both u and *qmap* depend on each other.

The above combinator allows us to write a code generator for the signature *LetSig* without resorting to an intermediate syntax tree. However, we have to be careful as this requires combining state transition functions with the same state space type: both *heightSt* and *scopeLvlSt* use the type *Int*.

However, the ambiguity can be easily resolved by "tagging" the types using *newtype* type synonyms. For the *scopeLvlSt* state transition, we define such a type like this:

```
newtype ScopeLvl = ScopeLvl { scopeLvl :: Int }
```

The tagging itself is a straightforward construction given the isomorphism between the type and its synonym in the form of a forward and a backward function:

```
tagDownState :: (q → p) → (p → q) → DownState f q → DownState f p
tagDownState i o t (q, s) = fmap i (t (o q, s))
```

We thus obtain a tagged variant of *scopeLvlSt*:

```
scopeLvlSt' :: ScopeLvlSt f ⇒ DownState f ScopeLvl
scopeLvlSt' = tagDownState ScopeLvl scopeLvl scopeLvlSt
```

The state maintained by *scopeLvlSt'* can now be accessed via the function *scopeLvl* in any compound state space containing *ScopeLvl*. A similar combinator can be defined for bottom-up state transitions.

Using the above state, we define a state transition function that assigns a memory address to each bound variable.

```
type VarAddr = Map Name Addr
class VarAddrSt f q where
  varAddrSt :: DDownState f q VarAddr
instance (ScopeLvl ∈ q) ⇒ VarAddrSt Let q where
  varAddrSt (LetIn v _ e) = e ↦ ((v ↦ scopeLvl above) ∪ above)
  varAddrSt _             = ∅
instance VarAddrSt f q where
  varAddrSt _ = ∅
```

Here, we use again overlapping instance declarations to give a uniform instance of *VarAddrSt* for all signatures different from *Let*.

We can now extend the type class *CodeSt* for the signature *Let*:

```
instance HeightSt Let where
  heightSt (LetIn _ x y) = 1 + max x y
  heightSt (Var _)       = 0
```

```
instance (ScopeLvl ∈ q, VarAddr ∈ q) ⇒ CodeSt Let q where
  codeSt (LetIn _ b e) = below b ++ [Store a] ++ below e
    where a = scopeLvl above
  codeSt (Var v)       = case lookup v above of
                           Nothing → error "unbound variable"
                           Just i  → [Load i]
```

Again, we have to be careful to avoid clashes in the use of addresses for storing intermediate results on the one hand and for storing results of let bindings on the other hand. Similar to our implementation in Section 6.1, we use the output of the bottom-up state transition to obtain the maximum address used for storing intermediate results.

Thus, we tie the different components of the computation together as follows:

```
codeLet′ :: Term LetSig → Code
codeLet′ t = c
  where (c, addr) = runDState (codeSt ⊗ dUpState heightSt)
                              (varAddrSt ⊛ dDownState scopeLvlSt′)
                              (∅ :: VarLvl, ScopeLvl (addr + 1)) t
```

Note that in both implementations, we could have avoided the use of the result of the state transition function *heightSt* to initialise the address counter for bound variables. The modularity of our recursion schemes makes it possible to replace the *heightSt* state transition function with a different one. In this way, we could avoid clashes by using even address numbers for intermediate results and odd address numbers for variables.

We already observed that stateful tree homomorphisms cannot discern the direction in which the state is propagated. Thus we can supply them with a state using either bottom-up or top-down state transitions. In fact, following the bidirectional state transitions we considered above, we can provide a stateful tree homomorphism with a combined state given by both a bottom-up and a top-down state transition function. Such a transformation can for example be used to rename apart all bound variables or inline simple let bindings.

7 Discussion

We have seen that with some adjustments tree automata can be turned into highly modular recursion schemes. These recursion schemes allow us to take advantage of two orthogonal dimensions of modularity: modularity in the state that is propagated and – courtesy of Swierstra's [23] *data types à la carte* – modularity in the structure of terms. In addition to that, we also showed how to decompose transducers into a homomorphism and into a state transition part. This high level of modularity makes our automata-based recursion schemes especially valuable for constructing modular compilers as we have illustrated in our running example. However, we should point out that there are many more aspects to consider when constructing compilers in a modular fashion [4].

The dependent forms of bottom-up and top-down state transitions that we have developed in this paper are nothing else than the *synthesised* and *inherited attributes* known from *attribute grammars* [22]. In fact, the combinator *runDState* that runs both a bottom-up and a top-down state transition can be seen as a run of an attribute grammar with corresponding synthesised and inherited attributes. Viera et al. [24] have developed a Haskell library that allows to specify such attribute grammars in Haskell in a very concise way.

We also obtain an added value by using a powerful functional language for the embedding of our recursion schemes. One immediate benefit that we obtain is the use of further generic programming techniques. For example, the *heightSt* state transition function could have been defined entirely generically, without having to extend the definition for every new signature.

Why Tree Transducers? In principle, tree transducers offer no increase in expressiveness over (dependent) bottom-up state transition functions since we allow for infinite state spaces anyway. However, due to their additional structure they provide at least two advantages.

First of all, tree transducers are very flexible in the way they can be manipulated in order to form new transformations. For example, we can extend a given signature functor f with annotations of some type a by using the construction

$$\textbf{data}\ (f :\&: a)\ e = f\ e :\&: a$$

A term over the signature $(f :\&: a)$ is similar to a term over f but it additionally contains annotations of type a at every subterm. We can provide a combinator that modifies a tree transducer from F to G into one from $F :\&: A$ to $G :\&: A$ that propagates the annotations from the input term to the output term [1].

Secondly, tree transducers can be composed. That is, given two bottom-up (respectively top-down) tree transducers – one from F to G, the other one from G to H, we can generically construct a bottom-up (respectively top-down) transducer from F to H whose transformation is equal to the composition of the transformations denoted by the original transducers [7]. The resulting transducer then only has to traverse the input term once and avoids the construction of the intermediate term [26]. Note that tree homomorphisms can be considered both a special case of bottom-up and of top-down tree transducers and can thus be composed with either kind.

The two abovementioned features also set tree transducers apart from other generic programming approaches such as *Scrap your Boilerplate* [13, 12, 17] or *Uniplate* [21]. We do not give the full technical details of the two features here but the implementation can be found in the `compdata` package [2].

Extensions & Future Work While we only considered single recursive data types, this restriction is not essential: following the construction of Yakushev et al. [27] and Bahr and Hvitved [1], our recursion schemes can be readily extended to work on mutually recursive data types as well.

Note that the *runDState* combinator of Section 6.2 constructs the product of the two state spaces u and d. Consequently, if u is a compound state space, we obtain a product type that is not a right-associative nesting of pairs which we require for the type class $\in$ to work properly. However, this can be remedied by a more clever encoding of compound state spaces as *heterogeneous lists* [14] or generating instance declarations for products of a limited number of components via *Template Haskell*.

The transducers that we have considered here have one severe limitation. This limitation can be seen when looking at the implementation of these transducers in Haskell: the parametric polymorphism of the type for placeholder variables prevents us from using these placeholder variables in the state transition. This would allow us to store and retrieve subterms that the placeholder variables are instantiated with. The ability to do that is necessary in order to perform "non-local" transformations such as inlining of arbitrary let bindings or applying substitutions. However, we can remedy this issue by making the state a functor. The type of bottom-up respectively top-down transducers would then look as follows:

$$\begin{aligned}&\textbf{type } \mathit{UpTrans} \quad f\ q\ g = \forall\ a\ .\ f\ (q\ a, a) \to (q\ (\mathit{Context}\ g\ a), \mathit{Context}\ g\ a)\\ &\textbf{type } \mathit{DownTrans}\ f\ q\ g = \forall\ a\ .\ (q\ a, f\ a) \to \mathit{Context}\ g\ (q\ (\mathit{Context}\ g\ a), a)\end{aligned}$$

We can then, for example, instantiate q with $\mathit{Map}\ \mathit{Var}$ such that the state is a substitution, i.e. a mapping from variables to terms (respectively term placeholders).

The above types represent a limited form of macro tree transducers [7]. Wile the decomposition of such an extended bottom-up transducer into a homomorphism and a state transition function is again straightforward, the decomposition of an extended top-down transducer is trickier: at least the representation with explicit placeholders that we used for dependent top-down state transition functions does not straightforwardly generalise to polymorphic states.

Note that the abovementioned limitation only affects transducers, not state transition functions. We can, of course, implement inlining and substitution as a bidirectional state transition. However, if we want to make use of the nice properties of transducers, we have to move to the extended tree transducers illustrated above.

Acknowledgements

The author would like to thank Tom Hvitved, Jeremy Gibbons, Wouter Swierstra, Doaitse Swierstra and the anonymous referees for valuable comments, corrections, suggestions for improvements and pointers to the literature.

Bibliography

[1] Bahr, P., Hvitved, T.: Compositional Data Types. In: Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming. pp. 83–94. WGP '11, ACM, New York, NY, USA (Sep 2011)

[2] Bahr, P., Hvitved, T.: Compdata (2012), `http://hackage.haskell.org/package/compdata`, module `Data.Comp.Automata`

[3] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2007)

[4] Day, L., Hutton, G.: Towards Modular Compilers For Effects. In: Proceedings of the Symposium on Trends in Functional Programming. Madrid, Spain (2011)

[5] Eilenberg, S., Wright, J.B.: Automata in general algebras. Inform. Control. 11(4), 452–470 (1967)

[6] Fokkinga, M.M.: Law and Order in Algorithmics. Ph.D. thesis, University of Twente, 7500 AE Enschede, Netherlands (1992)

[7] Fülöp, Z., Vogler, H.: Syntax-Directed Semantics: Formal Models Based on Tree Transducers. Springer-Verlag New York, Inc. (1998)

[8] Gibbons, J.: Upwards and downwards accumulations on trees. In: Bird, R., Morgan, C., Woodcock, J. (eds.) Mathematics of Program Construction. LNCS, vol. 669, pp. 122–138. Springer Berlin / Heidelberg (1993)

[9] Gibbons, J.: Polytypic downwards accumulations. In: Jeuring, J. (ed.) Mathematics of Program Construction. LNCS, vol. 1422, pp. 207–233. Springer Berlin / Heidelberg (1998)

[10] Gibbons, J.: Generic downwards accumulations. Sci. Comput. Program. 37(13), 37–65 (2000)

[11] Hasuo, I., Jacobs, B., Uustalu, T.: Categorical Views on Computations on Trees (Extended Abstract). In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (eds.) Automata, Languages and Programming. LNCS, vol. 4596, pp. 619–630. Springer Berlin / Heidelberg (2007)

[12] Hinze, R., Löh, A., Oliveira, B.: Scrap Your Boilerplate Reloaded. In: Functional and Logic Programming. pp. 13–29. Springer (2006)

[13] Hinze, R., Löh, A.: Scrap your boilerplate: revolutions. In: Proceedings of the Eighth International Conference on Mathematics of Program Construction,. LNCS, vol. 4014, pp. 180–208. Springer-Verlag (2006)

[14] Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell. pp. 96–107. ACM Press (2004)

[15] Knuth, D.E.: Semantics of context-free languages. Theory Comput. Syst. 2(2), 127–145 (1968)

[16] Kühnemann, A.: Benefits of Tree Transducers for Optimizing Functional Programs. In: Arvind, V., Ramanujam, S. (eds.) Foundations of Software

Technology and Theoretical Computer Science. LNCS, vol. 1530, p. 1046. Springer Berlin / Heidelberg (1998)
[17] Lämmel, R., Jones, S.P.: Scrap your boilerplate with class: extensible generic functions. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming. pp. 204–215. ACM, New York, NY, USA (2005)
[18] Lewis, J.R., Launchbury, J., Meijer, E., Shields, M.B.: Implicit parameters: dynamic scoping with static types. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 108–118. ACM, New York, NY, USA (2000)
[19] Marlow, S.: Haskell 2010 Language Report (2010)
[20] Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 523, pp. 124–144. Springer Berlin / Heidelberg (1991)
[21] Mitchell, N., Runciman, C.: Uniform boilerplate and list processing. In: Proceedings of the ACM SIGPLAN Workshop on Haskell. pp. 49–60. ACM, New York, NY, USA (2007)
[22] Paakki, J.: Attribute grammar paradigmsa high-level methodology in language implementation. ACM Comput. Surv. 27(2), 196–255 (Jun 1995)
[23] Swierstra, W.: Data types à la carte. J. Funct. Program. 18(4), 423–436 (2008)
[24] Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 245–256. ACM, New York, NY, USA (2009)
[25] Voigtländer, J.: Formal Efficiency Analysis for Tree Transducer Composition. Theory Comput. Syst. 41(4), 619–689 (2007)
[26] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Theor. Comput. Sci. 73(2), 231–248 (1990)
[27] Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 233–244. ACM, New York, NY, USA (2009)